Cascade Algorithm Revisited

Tadao Takaoka*, Kiyomi Umehara**
*Department of Computer Science
University of Canterbury
Christchurch, New Zealand
**Hitachi Laboratory
Tokyo, Japan

January 1990, revised November 2013

Abstract

We revisit the cascade algorithm for the all pairs shortest path (APSP) problem. The operation on the distace data is limited to the triple operation of $min\{a, b + c\}$. The best known complexity on this model is n^3 by Floyd's algorithm. The cascade algorithm takes $2n^3$ opearations. We first improve this bound to n^3, that is, on a par with Floyd's algorithm. Then we implement the improved version on a mesh computer and achieve $3n - 2$ communication steps.

1 Introduction

We consider a directed graph $G = (V, E)$ where $V = \{1, 2, ..., n\}$ is the set of vertices given by integers and E is the set of edges given by pairs of integers. We associate a non-negatice edge cost d_{ij} with edge (i, j). By setting d_{ij} to ∞ when pair (i, j) is not in E, we have an (n, n) square matrix $D = \{d_{ij}\}$. The seuqence of edges $(i, k_1)(k_1, k_2)...(k_m, j)$ where $m \geq 1$ is a path from i to j of length $m + 1$. Note that the path is just edge (i, j) when $m = 0$. The cost of the path is the sum of the costs of edges in the path. The cost of the shortest path from i to j is called the shortest distance from i to j. The all pairs shortest path (APSP) problem is to compute the shortest paths from i to j for all pairs (i, j). Let d^*_{ij} be the shortest distance from i to j, and $D^* = \{d^*_{ij}\}$. W will show how to compute D^* from the given matrix D. That is, we compute the shortest distance matrix D^* from D. D^* is called the closure of D. The shortest paths from i to j for all (i, j) can be computed as by-product in the process of computing D^*. Hence we mainly focus on how to compute D^*. We assume that the diagonal elements of D are 0.

There are many ways to compute D^*. See comprehensive reviews by Takaoka [9] and Zwick [10]. In this paper we focus on the so-clalled cascade alogorithm invented by Farby, Land and Murchland [2]. The correctness of the algorithm was later proved by Hu [4]. The algorithm is given below.

Note that previously computed d_{ij} are used later in the computation in this algorithm.

Classical cascade algorithm
{forward process}
for i:=1 **to** n **do for** i:=1 **to** n **do begin**
  c:=∞;
  for k:=1 **to** n **do** c:=min$\{c, d_{ik} + d_{kj}\}$
  d_{ij}:=c
end
{backward process}
for $i := n$ **downto** 1 **do for** $i := n$ **downto** 1 **do begin**
  c:=∞;
  for $k := n$ **downto** 1 **do** c:=min$\{c, d_{ik} + d_{kj}\}$
  d_{ij}:=c
end
for i:=1 **to** n **do for** i:=1 **to** n **do** $d^*_{ij} := d_{ij}$

In the forward process i, j and k sweep in increasing order, whereas in the backward process they sweep in decreasing order. Let us call the operation $\min\{a, b+c\}$ the triple operation where a, b and c are non-negative real numbers. In this paper we measure the complexity of algorithms by the number of triple operations executed. In the above cascade algorithm, the number of triple operations is obviously $2n^3$. In contrast, the following Floyd's algorithm [3] computes D^* with n^3 triple operations.

Floyd's algorithm
for k:=1 **to** n **do**
  for i:=1 **to** n **do for** j:=1 **to** n **do**
    d_{ij}:=min$\{d_{ij}, d_{ik} + d_{kj}\}$
for i:=1 **to** n **do for** i:=1 **to** n **do** $d^*_{ij} := d_{ij}$

Because of its simplicity and less complexity, Floyd's algorithm seems to be superior to the cascade algorithm and the latter seems to have been forgotten.

In the next section we improve the number of triple operations in the cascade algorithm to n^3. Johnson [7] showed that if only comparisons and additions are used in a straight-line program to solve the APSP problem, we need $2n(n-1)(n-2)$ operations. We show that both of Floyd's algorithm and the cascade algorithm are optimal in this computational model. A straight-line program has no branching on the processed data. We count only operations on distance data, not on control variables. In Sections 3 and 4, we give a correctness proof and analysis of the algorithm.

In Section 5, we further modify the improved cascade algorithm and show how to design a VLSI circuit for the modified cascade algorithm. The circuit is of $O(n^2)$ aize and takes $O(n)$ time. This is an improvement of the VLSI implementation by Sinha, et. al [**?**], which is of $O(n^2)$ size amd takes $O(n \log n)$ time and $O(n^2)$ propagaton time.

In Section 6, we show how to improve the cascade algorithm in average

case. The expected computing time of the improved version is $O(n^{2.5})$.

2 Improved cascade algorithm

A careful review of the proof of the algorithm in [4] brings us an improvement by limiting the sweeping range of the control variable k in both the forward and backward processes in the following way.

Improved cascade algorithm
{forward process}
for i:=1 **to** n **do for** j:=1 **to** n **do begin**
  c:=d_{ij};
  for k:=1 **to** $\min\{i,j\}-1$ **do** c:=$\min\{c, d_{ik}+d_{kj}\}$
  d_{ij}:=c
end
{backward process}
for i:=n **downto** 1 **do for** j:=n **downto** 1 **do begin**
  c:=d_{ij};
  for $k := n$ **downto** $\min\{i,j\}+1$ **do** c:=$\min\{c, d_{ik}+d_{kj}\}$
  d_{ij}:=c
end
for i:=1 **to** n **do for** i:=1 **to** n **do** $d^*_{ij} := d_{ij}$

We give the proof for the correctness in the next section.

3 Correctness

Let us denote the path $(i,k_1)(k_1,k_2)...(k_m,j)$ by vertices $(i,k_1,...,k_m,j)$.

DEFINITION 1 *The path $(i,k_1,...,k_m,j)$ is*
(1) an up sequence, if $i<k_1<...<k_m<j$,
(2) a down sequence, if $i>k_1>...>k_m>j$,
(3) a valley sequence, if $k_\ell<i$ and $k_\ell<j$) for all ℓ, or
(4) a hill sequence, if $k_\ell>i$ and $k_\ell>j$) for all ℓ.

We note that all verices in each of the above sequences are distinct. That is, we only consider simple paths. We use the fact that d_{ij} is computed later (earlier) than $d_{i'j'}$ is if $i'<i$ and $j'<j$ in the forward process (backward process). When $m=1$, that is, the path has only two edges, we call (3) a short valley sequence and (4) a short hill sequence.

LEMMA 1 *If the shortest path from i to j, $(i,k_1,...,k_m,j)$, is a valley sequence, the value of d_{ij} gives the shortest distance from i to j at the end of the forward process.*

Proof. Let the path length is the number of edges in the given path. Proof is by induction on the path length. To prove the basis we assume a short valley sequence (i,k_1,j) is the shortest path for $m=1$. Then edge (i,k_1) and edge (k_1,j) are the shortest paths from i to k_1 and from k_1 to j. Since they are the shortest distances and $k_1<\min\{i,j\}$, d_{ij} is given by $d_{ik_1}+d_{k_1j}$. See the Figure 1.

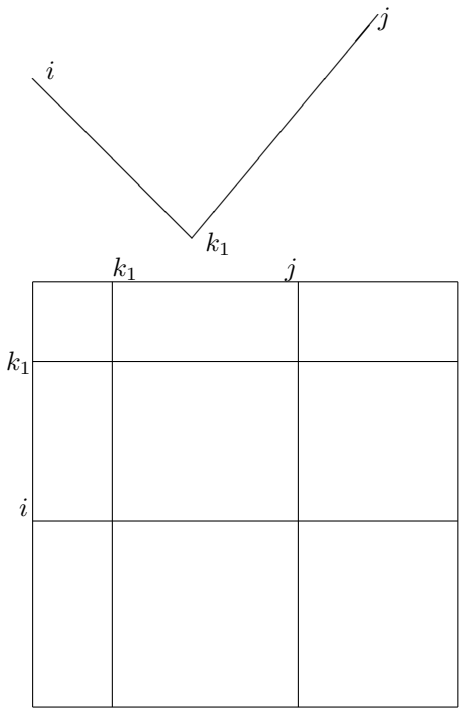

Figure 1: Short valley sequence and sweeping by k to captures k_1.

Now let $k_\ell = \max\{k_1, ..., k_m\}$, $(m > 1)$, that is, the path length is $m+1$. Then the sequence $(i, k_1, ..., k_\ell)$ and $(k_\ell, ..., k_m, j)$ form valley sequences, or single edges when $\ell = 1$ or $\ell = m$. By the induction hypothesis as their path lengths are up to m, we can assume d_{ik_ℓ} and $d_{k_\ell j}$ give the shortest distances from i to k_ℓ and the shortest distance from k_ℓ to j. Since $k_\ell < \min\{i, j\}$, d_{ij} is given by $d_{ik_\ell} + d_{k_\ell j}$. See Figure 2.

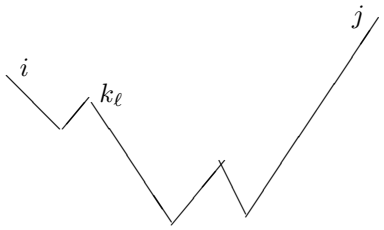

Figure 2: Long valley sequence

If the algorithm computes the shortest distance from i to j in d_{ij}, we assume a hypothetical edge (i, j) with cost d_{ij}. We say the shortest path from i to j has been reduced to edge (i, j). If the original edge (i, j) is the shortest, the algorithm keeps it intact.

Now we modify the forward process in the following way.We call the modified process the long forward process (LFP). In contrast, we call the original forward process the short forward process (SFP). The naming comes from the distance of sweeping by the control variable k.

{Long forward process}
for i:=1 **to** n **do for** j:=1 **to** n **do begin**
  c:=d_{ij};
  for k:=1 **to** $\max\{i,j\}-1$ **do** c:=$\min\{c, d_{ik}+d_{kj}\}$
  d_{ij}:=c
end

LEMMA 2 *If the shortest path from i to j, $(i, k_1, ..., k_m, j)$, is one of the following three, the value of d_{ij} gives the shortest distance from i to j at the end of the long forward process. We define an extended valley sequnce as a vally sequence preceded by an initial down sequence or followed by a final up sequence. An extended hill sequence is defined similarly.*

(1) Extended valley sequence
(2) Up sequence
(3) Down sequence.

Proof. For (1), let us assume an extended valley sequence from i to j such that $i < j$. Let k be the minimum of vertices on the final up-sequence that is greater than i. The subsequence from i to k is a valley sequence covered by Lemma 1, which reduces the valley sequence into a single edge and forms an up-sequence with the sequence from k to j. Since the vertices from k to j appear in increasing order in the algorithm, the computational process is the same as for an up-sequence in (2). The case of $i > j$ is symmetric.

We prove (2) by induction. To prove the basis we assume an up-sequence is an edge(i, j). Obviously d_{ij} gives the shortest distance at the end of the long forwards process. Assume by induction that d_{ik_m} gives the shortest distance from i to k_m after the LFP. Then the shortest distance d_{ij} from i to j is given by $d_{ik_m} + d_{k_m j}$, since $k_m < \max\{i, j\}$. See Figure 3. Note that the distances on the path $(i, k_1, ..., k_m, j)$ are computed in the order of $d_{ik_1}, ..., d_{ik_m}, d_{ij}$. The proof for (3) is is similar and seen from Figure 4. Note that the shortest distances on the path $(i, k_1, ..., k_m, j)$ are computed in the order of $d_{k_m j}, ..., d_{k_1 j}, d_{ij}$.

Now we call the original backward process the long backward process (LBP), and define the short backward process (SBP) in the following.

{Short backward process}
for i:=n **downto** 1 **do for** j:=n **downto** 1 **do begin**
  c:=d_{ij};
  for $k := n$ **downto** $\max\{i,j\}+1$ **do** c:=$\min\{c, d_{ik}+d_{kj}\}$
  d_{ij}:=c
end

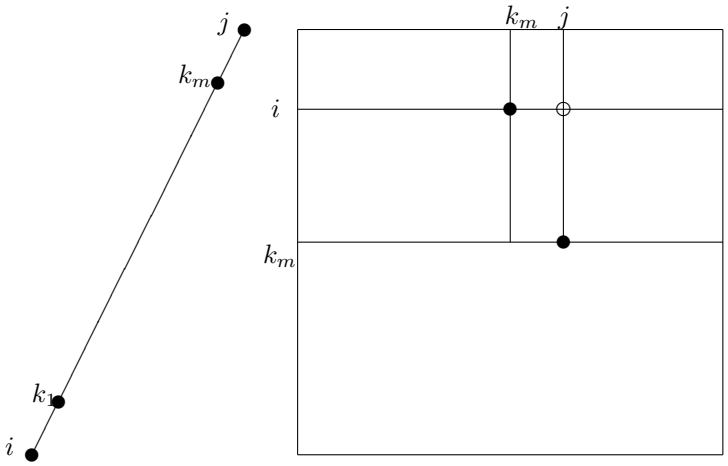

Figure 3: Working on the up-sequence

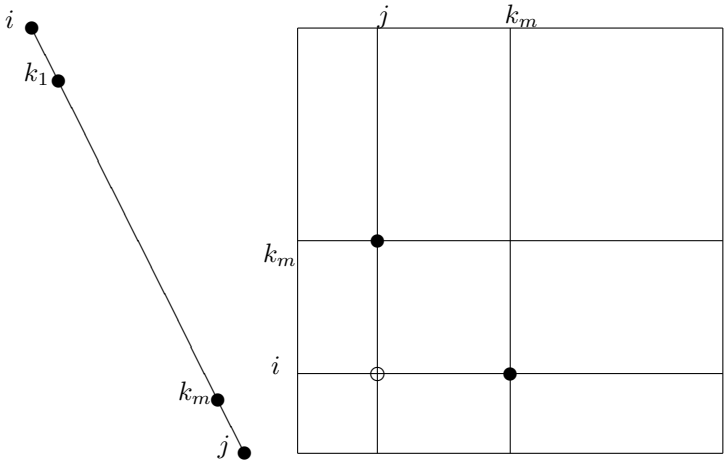

Figure 4: Working on the down-sequence

LEMMA 3 *If the shortest path from i to j, $(i, k_1, ..., k_m, j)$, is a hill sequence, the value of d_{ij} gives the shortest distance from i to j at the end of the short backward process.*

Proof. Similar to Lemma 1.

LEMMA 4 *If the shortest path from i to j, $(i, k_1, ..., k_m, j)$, is one of the following three, the value of d_{ij} gives the shortest distance from i to j at the end of the long backward process.*

(1) Extended hill sequnce
(2) Up sequence
(3) Down sequence.

Proof. Similar to Lemma 2.

Note that in Lemma 3 and Lemma 4, we execute the SBP and LBP in isolation, not in conjunction with a forward process. Now we define by (P, Q) the execution of the processes P and Q in this order. The first case of the following theorem is the improved cascade algorithm we mentioned earlier in the paper.

THEOREM 1 *The execution of each of (SFP, LBP) and (SBP, LFP) computes in d_{ij} the shortest distance from i to j.*

proof. We give a proof for (SFP, LBP). The other proof is similar. Now we reduce the shortest path from i to j, $(i, k_1, ..., k_m, j)$ to $(i, a_1, ..., a_r, j)$ by reducing the maximal valley parts into edges. See Figure 5. Vertices $a_1, ..., a_r$ are such that the sub-sequence from a_k to a_{k+1} forms a maximal valley sequence, an up-sequence or a down-sequence. A valley sequece is maximal if it is a valley sequence and it is not a part of a larger valley sequence. The sequence $(i, a_1, ..., a_r, j)$ is in general a composite of an up-sequence followed by a down-sequence. Obviously the reduced sequence $(i, a_1, ..., a_r, j)$ is an extended hill sequence, an up-sequence or a down-sequence. The SFP computes the shortest distance between any pair of vertices in the reduced sequence. By assuming hypothetical single edges with distances obtained in the SFP between these pairs, we see that the LBP computes the shortest distances from i to j.

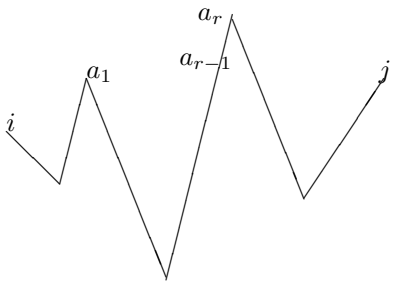

Figure 5: General sequence

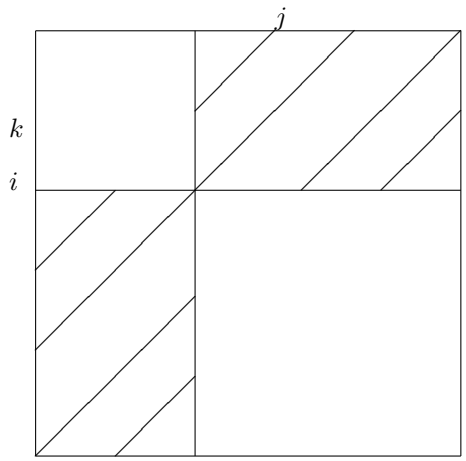

Figure 6: Analysis of SFP

4 Analysis

We count the number of triple operatons in the improved cascade algorithm. Note that a triple operation consists of one comparison and one addition. In the SFP, the number is measured by the summation of the hatched part in Figure 6. The number of triple operationd is given by

$$F = 2\Sigma_{i=1}^{n}(i-1)(n-i) = 2(n+1)\Sigma_{i=1}^{n} i - 2\Sigma_{i=1}^{n} i^2 - 2n^2$$

$$= (n+1)n(n+1) - (2/6)n(n+1)(2n+1) - 2n^2$$

$$= (1/3)n^3 - n^2 + (2/3)n$$

We analyze the LBP by using the LFP, since the two are computaionally symmetric. See Figure 7. The number of triple operations is

$$B = F + \Sigma_{i=1}^{n}[(n-i)^2 - (n-i)] = F + (1/3)n^3 - n^2 + (2/3)n$$

Hence the total number of triple operations is given by $F + B = n^3 - 3n^2 + 2n$. In this analysis we changed the LFP slightly so that k skips the cases $k = i$ if $i < j$ and $k = j$ if $i > j$. In Theorem 2, this changed version is used.

In the k-th iteration of the Floyd algorithm, we can avoid updating the value of d_{ij} in the k-th row and column, and also we can avoid updating diagonal elements. In this version, the Floyd algorithm takes $n(n-1)(n-2)$ triple operations, which is equal to the above figure. Also in Johnson, is is shown that $2n(n-1(n-2)$ operations are required if the computational model is limited to the one introduced in Introduction. Hence we have

THEOREM 2 *The improved cascade algorithm is optimal in the computational model of straight-line program with comparisons and additions.*

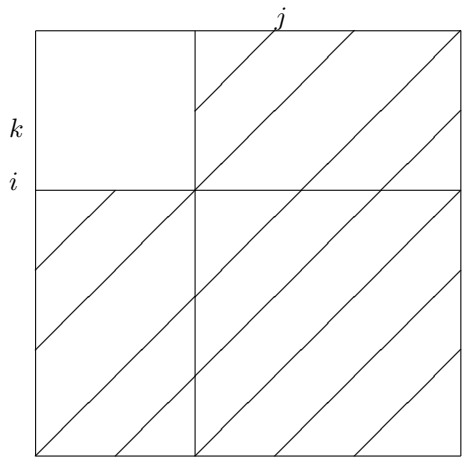

Figure 7: Analysis of LFP

Now we try to speed up the algorithm for acyclic graphs. We suppose that the given graph is acyclic and vertices are topologically sorted, that is, if $(i, j) \in E$ then $i < j$. Then the shortest paths are all up sequences. Hence the shortest distances are obtained afte the LFP. But we can speed up the LFP even more in the following.

{Modified long forward process, MLFP}
for i:=1 **to** n **do for** $j := i + 1$ **to** n **do begin**
  c:=d_{ij};
  for $k := i + 1$ **to** $j - 1$ **do** c:=min$\{c, d_{ik} + d_{kj}\}$
  d_{ij}:=c
end

An easy calculation reveals that the number of triple operations is $(1/6)n(n-1)(n-2)$. This figure is equal to that for the Floyd algorithm if it is applied to an upper triangle matrix for an acycle graph.

If vertices are toplogically sorted in reverse order, the following algorithm computes the shortest distances.

{Modified long backward process}
for i:=n **downto** 1 **do for** $j := i - 1$ **downto** 1 **do begin**
  c:=d_{ij};
  for k:=$i - 1$ **downto** $j + 1$ **do** c:=min$\{c, d_{ik} + d_{kj}\}$
  d_{ij}:=c
end

Note that MLFP and MLBP are computationally symmetric. The former works on the upper-right triangle and the latter on the lower-left triangle of the matrix.

Lemma 5 *MLFP and MLBP computes shortest distances for an up se-*

quence and a down sequence respectively.
proof *Let $(i, k_1, ..., k_m, j)$ be the up sequence that gives shortest path from i to j. From induction we can assume the shortest distance for the up sequence $(i, k_1, ..., k_m)$ is computed in d_{ik_m}. Since $i < k_m < j$, the MLFP captures the shortest path successfully. Similarly the MLBP computes the shortest distance for the down sequence.*

From discussions from the previous sections, we have the following theorem useful for VSLI implementation.

THEOREM 3 *The sequential execution of (SFP, SBP, MLFP, MLBP) in this order computes all shortest distances. The number of triple operations is $n(n-1)(n-2)$, which is optimal in the same computational model as in the previous section.*
Proof *The SFP reduces the shortest path from i to j, $(i, k_1, ..., k_m, j)$, to $(i, a_1, ..., a_r, j)$ as described in Theorem 1. This is:*
(1) an up-sequence,
(2) a down sequence, or
(3) an up sequence followed by a down sequence, that is, extended hill sequence.

In case of (3), the SBP reduces it into an up sequence or down sequence. The MFLP computes the shortest distance for the up sequences and the MLBP computes the shortest distances for the down sequences, from Lemma 5. The analysis is straightforward.

5 Approach by forward process only

LEMMA 6 *LFP reduces the shortest path sequence from i to j to an up-sequence, a down sequence, or an extended hill sequence. LBP reduces the shortest path sequence from i to j to an up-sequence,a down-sequence, or an extended valley sequence.*
Proof. *LFP reduces an extended valley sequence from i to j to an edge going up if $i < j$ and going down if $i > j$. The latter half of the lemma is symmetric.*

Let us square the given distance matrix three times based on the LFP given in the previou sections. Then we have:

THEOREM 4 *Two LFP squarings and a full squaring solve the APSP problem. The number of triple operations is $2n^3 - 6n^2 + 4n$.*
Proof *The first squaring reduces the shortest paths into an up-sequence, down-sequence or extended hill sequence. The second squaring reduces the up-sequences and down-sequences into single edges, leaving only short hill sequences. The last squaring computes shortest distances for those short hill sequences. For the analysis, the number of triple operations is $3((2/3)n^3 - 2n^2 + (4/3)n) = 2n^3 - 6n^2 + 4n$.*

There are two methods for implementing APSP algorithms on a VLSI. One is to have the input buffer in addition to the main mesh architecture, such that the input distances are prepared in the buffer. The other is based on the assumption that the input dstances are already stored in the

mesh architecture. We call the first the buffer method and the second the in-place method. We begin with the buffer method.

The input is the skewed and reversed matrix fed into the mesh from left and the same matrix skewed and reversed vertically, fed from above vertically. Thus the mesh architecture square the given distance matrix with cascading. Ignoring the constant terms, direct implementation of matrix multiplication on VLSI takes $3n$ steps. We showed three LFP multiplications solve APSP. If we use wrap around connection APSP can be solved in $5n$ steps. As Umehara points out, if we feed the mesh from four directions, and take the final result from the upper triangle and lower triangle, this reduces to $4n$ (Umehara's Masters thesis at Ibaraki University, 1990). On the other hand Bae, Shinn and Takaoka invented a mesh algorithm for matrix multiplication that takes $3.5n$ steps. That can be applied to our APSP problem, resulting in $3.5n$ steps on a VLSI, which is on a par with the best bound by Takaoka and Umehara in 1992.

Let R_{ij} be the accumulator in $cell(i,j)$. Let d_{ij} and d'_{ij} be the distance data in the input buffer, the former go from left to right and the latter from top to down. The algorithm at $cell(i,j)$ is described below.

Algorithm for $cell(i,j)$ for LFP
1. Receive the new values of d and d' from left and up.
2. If d_{ij} arrives at $cell(i,j)$, perform d_{ij}:=R_{ij} (Data release).
3. If d'_{ij} arrives at $cell(i,j)$, perform d'_{ij}:=R_{ij} (Data release).
4. At time $i+j+k-2$, R_{ij}:=min$\{R_{ij}, d_{ik}+d'_{kj}\}$ is performed

At $cell(i,j)$, the triple operation R:=min$\{R, d+d'\}$ is executed using R, d, d' registers. In step 4, i, j and k are meant to be comments. Note that if d_{ij} or d'_{ij} does not arrive, the d or d' value is sent unchanged to the next cell. Also note that R_{ij} is sent right or down before its completion in general, that is, k sweeps from 1 to max$\{i,j\}$-1, not to n. In the last squaring, the value of d_{ij} and d'_{ij} released from R_{ij}. The last R_{ij} is obtained by sweeping k to n beyond the release time.

LEMMA 7 *The distance d_{ij} arrives at $cell(i,k)$ at time $i+j+k-2$. Similarly d'_{ij} arrives at $cell(k,j)$ at time $i+j+k-2$.*
Proof. *Note that the sums of two indices are equal in each column of the left buffer. The sum of those in the column that includes d_{ij} is $i+j-1$. The column arrives at the left border of the mesh at $i+j-1$. Then to reach $cell(i,k)$, it takes another $k-1$ steps. The proof for d'_{ij} is similar.*

LEMMA 8 *Both d_{ik} and d'_{kj} arrive at $cell(i,j)$ at the same time $i+j+k-2$.*
Proof. *In the previous lemma, change the roles of k and j and k and i.*

This lemma ensures that the the two distance data meet at the right place at the right time in the step 1 of the algorithm.

LEMMA 9 *Each $cell(i,j)$ can correctly perform d_{ij}:=R_{ij} at time $i+2j-2$ and d'_{ij}:=R_{ij} at time $2i+j-2$.*
Proof. *The sweeping by k stops at* max$\{i,j\}-1$ *in LFP. The partially completed value of c must be consumed down stream in row i and column j. Line 2 and 3 in the above algorithm ensures the value made by the sweeping by k is consumed down stream. From the previous lemma, d_{ij}*

arrives at cell(i,j) at $i+2j-2$ and d'_{ij} arrives at cell(i,j) at $2i+j-2$. We prepare control signals. Both originates at cell$(1,1)$ at time 1. The first signal goes down at speed 1 and upon receiving, each cell sends a signal to the right at speed 1/2. The second signal is a mirror image of the first signal.

Note. Another way for the timing of data release is to prepare control signals 1 to k, and use the main diagonal for the origin of each control signal. The k-th control signal originates at $cell(k,k)$ at time $3k-2$ and propagates right and down spending one unit of time from cell to cell. Then it propagates to $cell(i,j)$ at time $3k-2+i-k+j-k = i+j+k-2$. For $i < j$, the j-th control signal comes to $cell(i,j)$ at time $i+2j-2$ and for $i > j$, the i-th control signal comes to $cell(i,j)$ at time $2i+j-2$.

THEOREM 5 *The algorithm is correct. It finishes at $k = 5n-2$.*

If we perform the cascade multiplication 3 times, the first two take n steps each. The last takes $3n-2$ steps, resulting in $5n-2$ steps altogether. The computation of the three multiplications go in a pipe-line fashion. The input is loaded into the buffers initially. The first multiplication takes n steps to bring d_{11} to the right border. The right border is connected to the left border by wrap arounds. As soon as a datum comes to the right border, it is sent to the left border, effectively starting the second multiplication with the control signal starting at $cell(1,1)$ again. As the values of d_{ij} and d'_{ij} go through the mesh in skewed forms, the data are fed into the mesh in skewed forms after n steps. Then as soon as the second multiplication spends n steps, the third starts with inputs again through the wrap arounds in skewed forms, taking $3n-2$ steps this time.

THEOREM 6 *APSP can be solved on a mesh array in $4n$ steps.*

Proof We prepare skewed matrices to be fed from left, from up, from right and from down. The computation on the data from left and top can be done in the first layer and that on the data from right and down can be done in the second layer. The above described computation can stop at $4n-1$ steps, the last multiplication taking $2n-1$ steps. The two computations go independently in the two layers, the second computation being a mirror image of the first. The result can be obtained from the top-left triangle of the first layer and the bottom-right triangle of the second layer, spending one more step.

For the lower bound, take an example graph, $G = (V,E)$ where $E = \{(1,n),(n,2),(2,n-1)\}$. Obviously the shortest distance from vertex 1 to vertex $n-1$ is $d_{1,n}+d_{n,2}+d_{2,n-1}$. We have four copies of $d_{n,2}$ in the input buffers. The distance $d_{n,2}$ needs to traverse from its original positions to $cell(1,n-1)$ taking $2n-2$ steps. Thus ignoring constant terms, we still have the gap of $2n$ steps.

References

[1] Sung Eun Bae and Tadao Takaoka, Algorithms for the Problem of K Maximum Sums and a VLSI Algorithm for the K Maximum Subarrays Problem. I-SPAN 2004: 247-253 (2004)

[2] B. A. Farby, A. H. Land J. D. Murchland, The cascade algorithm for finding all shortest distances in a directed graph, Management Science 14, 19-28, 1967

[3] Floyd, R. W. Algorithm 97: Shortest path. Comm. ACM. 5,6 (June 1962), p. 345.

[4] Hu, T. C. Revised matrix algorithm for shortest paths. SIAM J. Appl. Math. 15, I (Jan. 1967), 207-218.

[5] Lakhani, G., and Dorairaj, R. A VLSI implementation of all-pair shortest path problem. Proc. 1987 International Conference on Parallel Processing. IEEE Computer Society, 1987, pp. 207-209.

[6] Lewis, P., and Kung, S. An optimal systolic array for the algebraic path problem. IEEE Trans. Comput. C-40, 1 (Jan. 1991), 100-105.

[7] D. B. Johnson, Algorithms for shortest paths, TR-73-169, Ph. D thesis, dept of Computer Science, Cornell University, 1973 .

[8] van de Snepscheut, J. L. A. A derivation of a distributed implementation of Warshall s algorithm. Sci. Comput. Programming. 7, 2 (Sept. 1986), 55-60.

[9] Tadao Takaoka: All Pairs Shortest Paths via Matrix Multiplication. Encyclopedia of Algorithms 2008

[10] Uri Zwick: Exact and Approximate Distances in Graphs - A Survey. ESA 2001: 33-48